Searching

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

2-3 search trees: Toward balanced binary search trees

Balanced tree: any path from the root to a leaf has length $\lceil \log_2(N+1) \rceil$ (in terms of the number of nodes on the path).

2-3 search trees: Toward balanced binary search trees

Balanced tree: any path from the root to a leaf has length $\lceil \log_2(N+1) \rceil$ (in terms of the number of nodes on the path).

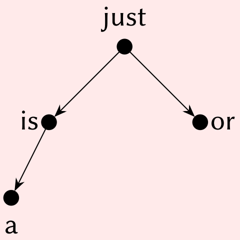
Maintaining *perfect* balance during additions and removals sounds highly expensive: Do we have to check the lengths of all paths and correct?

2-3 search trees: Toward balanced binary search trees

Balanced tree: any path from the root to a leaf has length $\lceil \log_2(N+1) \rceil$ (in terms of the number of nodes on the path).

Maintaining *perfect* balance during additions and removals sounds highly expensive: Do we have to check the lengths of all paths and correct?

A *balanced* binary search tree with $N = 4$ nodes



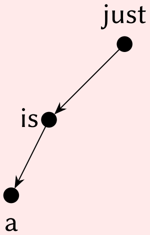
Consider removing "or".

2-3 search trees: Toward balanced binary search trees

Balanced tree: any path from the root to a leaf has length $\lceil \log_2(N+1) \rceil$ (in terms of the number of nodes on the path).

Maintaining *perfect* balance during additions and removals sounds highly expensive: Do we have to check the lengths of all paths and correct?

A *balanced* binary search tree with $N = 4$ nodes



Consider removing “or”: *paths are now too long!*

2-3 search trees: Toward balanced binary search trees

2-3 search trees: Toward balanced binary search trees

With a bit of flexibility, we can keep trees balanced-enough when adding or removing v by *only* making changes *locally* along a path from root to the node holding v.

2-3 search trees: Toward balanced binary search trees

With a bit of flexibility, we can keep trees balanced-enough when adding or removing v by *only* making changes *locally* along a path from root to the node holding v.

2-3 search trees

In a 2-3 tree, there are two types of nodes:

Two-nodes that hold one key value k and two children l and r.

Three-nodes that hold two key values k_1, k_2 and three children c_0, c_1, and c_2.

Furthermore, all leaf nodes in a 2-3 tree must have the *same* distance to the root.

2-3 search trees: Toward balanced binary search trees

With a bit of flexibility, we can keep trees balanced-enough when adding or removing v by *only* making changes *locally* along a path from root to the node holding v.

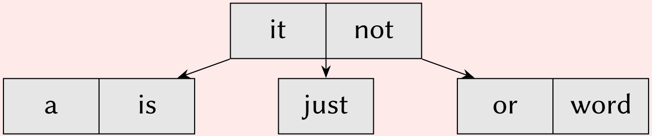
2-3 search trees

In a 2-3 tree, there are two types of nodes:

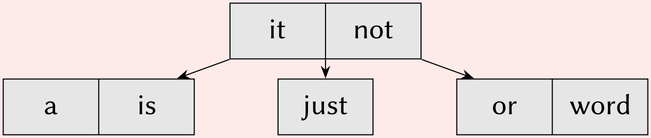
Two-nodes that hold one key value k and two children l and r.
l holds values $< k$ and r holds values $> k$.

Three-nodes that hold two key values k_1, k_2 and three children c_0, c_1, and c_2.
c_0 holds values $< k_1$, c_1 holds values $> k_1, < k_2$, and c_2 holds values $> k_2$.

Furthermore, all leaf nodes in a 2-3 tree must have the *same* distance to the root.

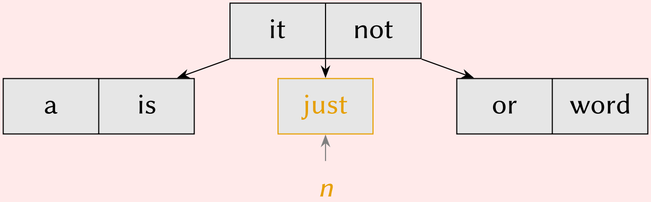


2-3 search trees: Toward balanced binary search trees



Consider adding "juice", "bee", and "zoo"

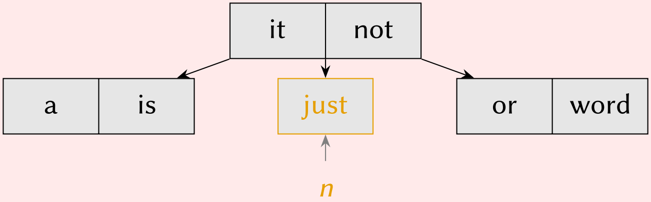
2-3 search trees: Toward balanced binary search trees



Consider adding “juice”, “bee”, and “zoo”

1. Search for “juice”: we find the leaf *two-node* n holding “just”.

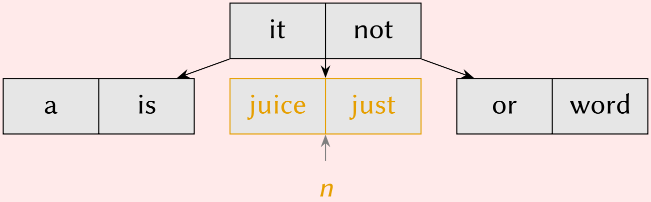
2-3 search trees: Toward balanced binary search trees



Consider adding “juice”, “bee”, and “zoo”

1. Search for “juice”: we find the leaf *two-node* *n* holding “just”.
2. We can turn node *n* into a *three-node*.

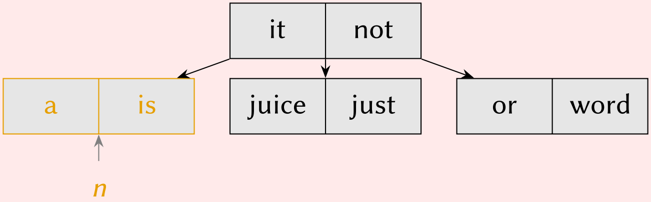
2-3 search trees: Toward balanced binary search trees



Consider adding “juice”, “bee”, and “zoo”

1. Search for “juice”: we find the leaf *two-node* *n* holding “just”.
2. We can turn node *n* into a *three-node*.

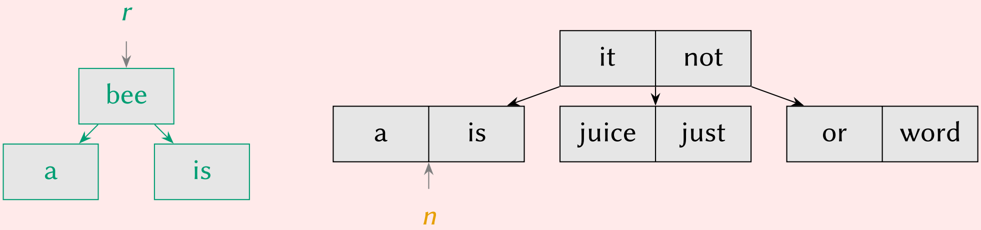
2-3 search trees: Toward balanced binary search trees



Consider adding "juice", "bee", and "zoo"

1. Search for "bee": we find the leaf *three-node* *n* holding "a" and "is".

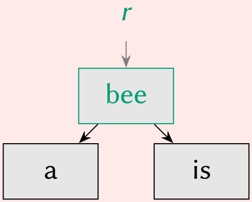
2-3 search trees: Toward balanced binary search trees

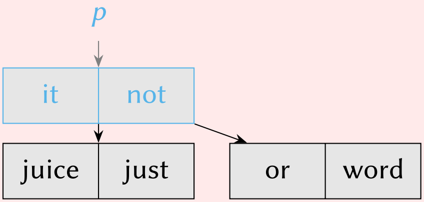


Consider adding “juice”, “bee”, and “zoo”

1. Search for “bee”: we find the leaf *three-node* n holding “a” and “is”.
2. We can turn the key values “a”, “bee”, “is” into a tree of two-nodes with root r.

2-3 search trees: Toward balanced binary search trees

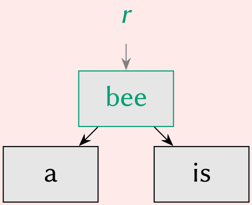


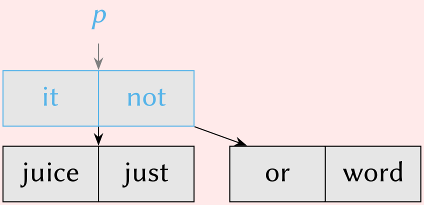


Consider adding “juice”, “bee”, and “zoo”

1. Search for “bee”: we find the leaf *three-node* n holding “a” and “is”.
2. We can turn the key values “a”, “bee”, “is” into a tree of two-nodes with root r.
3. Remove n and merge root r with the parent p of n:
   the merged node will have three key values “bee”, “it”, “not”.

2-3 search trees: Toward balanced binary search trees

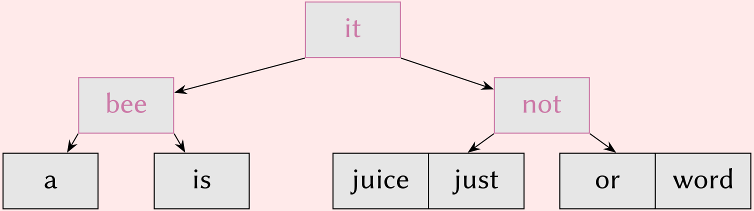




Consider adding "juice", "bee", and "zoo"

1. Search for "bee": we find the leaf *three-node* n holding "a" and "is".
2. We can turn the key values "a", "bee", "is" into a tree of two-nodes with root r.
3. Remove n and merge root r with the parent p of n:
   the merged node will have three key values "bee", "it", "not".
4. Represent these keys by a tree of two-nodes.

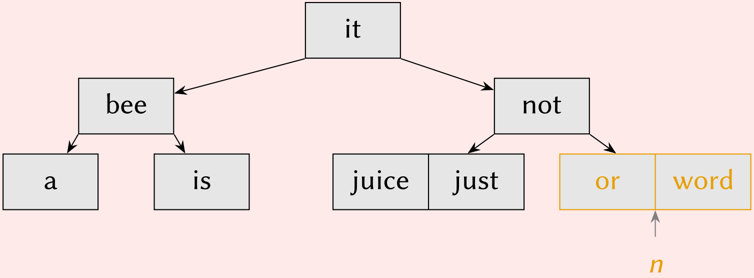
2-3 search trees: Toward balanced binary search trees



Consider adding "juice", "bee", and "zoo"

1. Search for "bee": we find the leaf *three-node* n holding "a" and "is".
2. We can turn the key values "a", "bee", "is" into a tree of two-nodes with root r.
3. Remove n and merge root r with the parent p of n:
   the merged node will have three key values "bee", "it", "not".
4. Represent these keys by a tree of two-nodes.

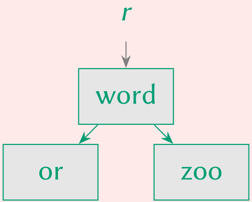
2-3 search trees: Toward balanced binary search trees

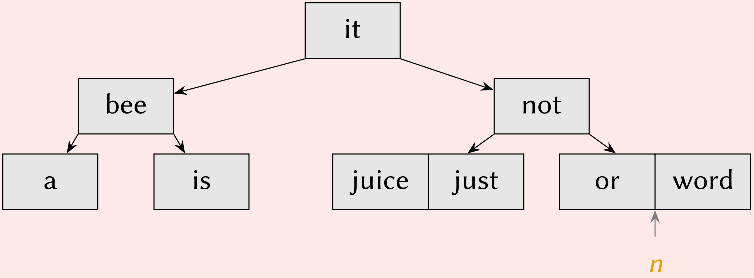


Consider adding “juice”, “bee”, and “zoo”

1. Search for “zoo”: we find the leaf *three-node n* holding “or” and “word”.

2-3 search trees: Toward balanced binary search trees

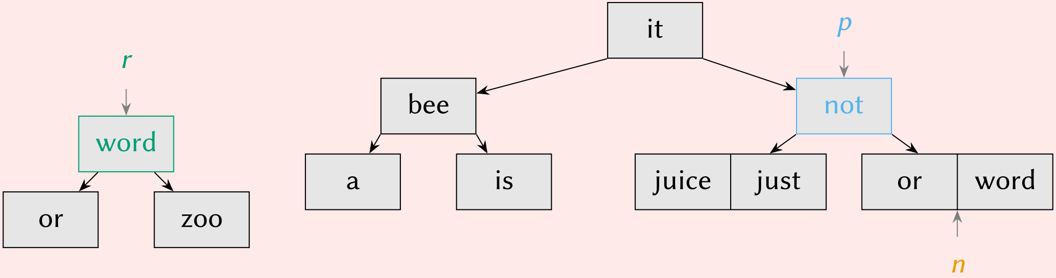




Consider adding "juice", "bee", and "zoo"

1. Search for "zoo": we find the leaf *three-node* n holding "or" and "word".
2. We can turn the key values "or", "word", "zoo" into a tree of two-nodes with root r.

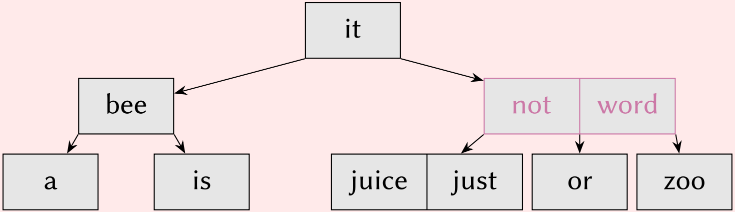
2-3 search trees: Toward balanced binary search trees



Consider adding “juice”, “bee”, and “zoo”

1. Search for “zoo”: we find the leaf *three-node* n holding “or” and “word”.
2. We can turn the key values “or”, “word”, “zoo” into a tree of two-nodes with root r.
3. Remove n and merge root r with the parent p of n:
   the merged node will have two key values “not” and “word”.

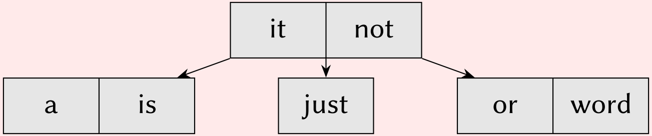
2-3 search trees: Toward balanced binary search trees



Consider adding "juice", "bee", and "zoo"

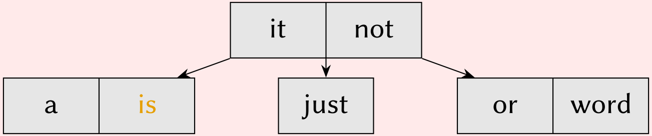
1. Search for "zoo": we find the leaf *three-node* n holding "or" and "word".
2. We can turn the key values "or", "word", "zoo" into a tree of two-nodes with root r.
3. Remove n and merge root r with the parent p of n:
   the merged node will have two key values "not" and "word".

2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

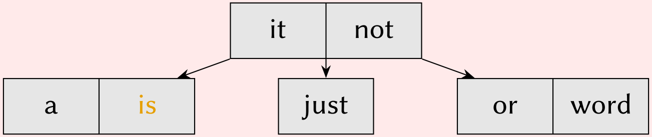
2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

1. Deleting a value from a leaf three-node *n*.

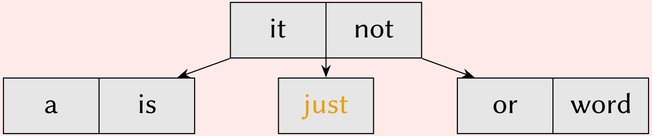
2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

1. Deleting a value from a leaf three-node *n*.
   Straightforward: replace the node *n* by a two-node.

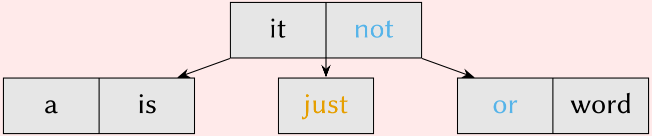
2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

1. Deleting a value from a leaf three-node *n*.
   Straightforward: replace the node *n* by a two-node.
2. Deleting a value from a leaf two-node.

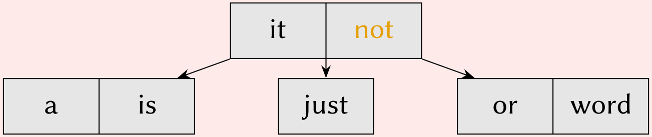
2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

1. Deleting a value from a leaf three-node *n*.
   Straightforward: replace the node *n* by a two-node.
2. Deleting a value from a leaf two-node.
   Complex: borrow an adjacent value from the parent, *recursively* if the parent itself is a two-node.

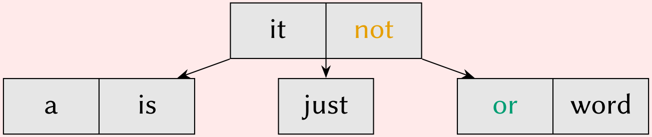
2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

1. Deleting a value from a leaf three-node *n*.
   Straightforward: replace the node *n* by a two-node.
2. Deleting a value from a leaf two-node.
   Complex: borrow an adjacent value from the parent,
   recursively if the parent itself is a two-node.
3. Deleting an internal value.

2-3 search trees: Toward balanced binary search trees



Deleting values from a 2-3 tree

1. Deleting a value from a leaf three-node *n*.
   Straightforward: replace the node *n* by a two-node.
2. Deleting a value from a leaf two-node.
   Complex: borrow an adjacent value from the parent,
   recursively if the parent itself is a two-node.
3. Deleting an internal value.
   Complex: replace value by the succeeding value (a leaf value),
   remove that leaf value.

2-3 search trees in practice

2-3 trees have *at least* two children per internal node:

2-3 trees can be *compacter* (in their height) than balanced search trees!

2-3 search trees in practice

2-3 trees have *at least* two children per internal node:
2-3 trees can be *compacter* (in their height) than balanced search trees!

2-3 trees require *complex* tree algorithms, however:
e.g., separate code to deal with two-nodes and three-nodes.

2-3 search trees in practice

2-3 trees have *at least* two children per internal node:
2-3 trees can be *compacter* (in their height) than balanced search trees!

2-3 trees require *complex* tree algorithms, however:
e.g., separate code to deal with two-nodes and three-nodes.

2-3 trees are *costly* for very large values:
when adding or removing values, other values are moved around in memory!

2-3 search trees in practice

2-3 trees have *at least* two children per internal node:
2-3 trees can be *compacter* (in their height) than balanced search trees!

2-3 trees require *complex* tree algorithms, however:
e.g., separate code to deal with two-nodes and three-nodes.

2-3 trees are *costly* for very large values:
when adding or removing values, other values are moved around in memory!

2-3 trees can be generalized to $(k - 2k)$-trees that are even compacter:
these $(k - 2k)$-trees are at the basis of external memory data structures,
e.g., B+trees that are widely used in file systems and large-scale databases.

From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

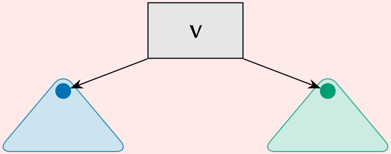
Idea: Turn 2-3 tree nodes into binary search tree structures.

From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

Idea: Turn 2-3 tree nodes into binary search tree structures.

- Two-nodes are already binary search tree nodes.

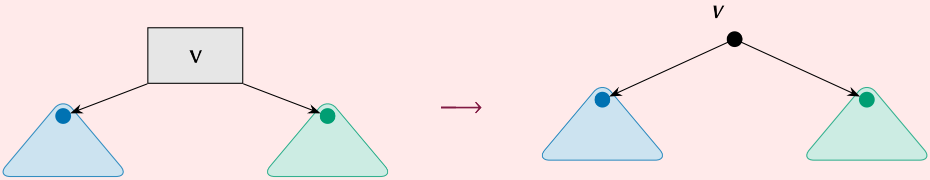


From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

Idea: Turn 2-3 tree nodes into binary search tree structures.

- Two-nodes are already binary search tree nodes.

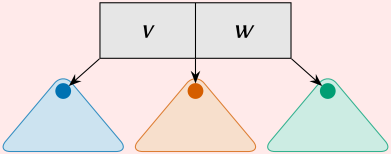


From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

Idea: Turn 2-3 tree nodes into binary search tree structures.

- Two-nodes are already binary search tree nodes.
- Three-nodes can be replaced by a binary search tree structure with two nodes.

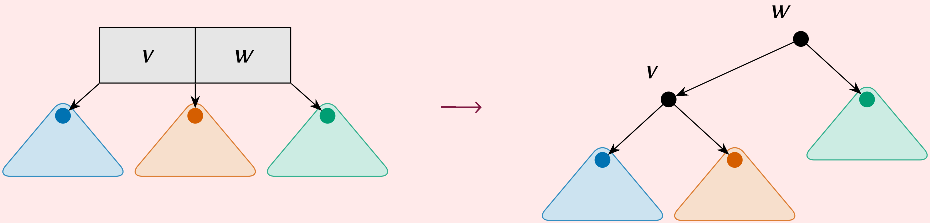


From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

Idea: Turn 2-3 tree nodes into binary search tree structures.

- Two-nodes are already binary search tree nodes.
- Three-nodes can be replaced by a binary search tree structure with two nodes.

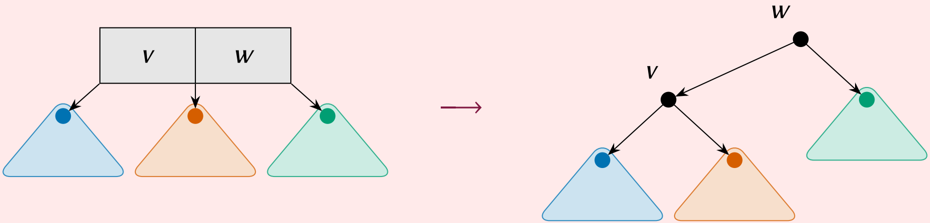


From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

Idea: Turn 2-3 tree nodes into binary search tree structures.

- Two-nodes are already binary search tree nodes.
- Three-nodes can be replaced by a binary search tree structure with two nodes.



Reusing the addition and removal algorithms from 2-3 trees

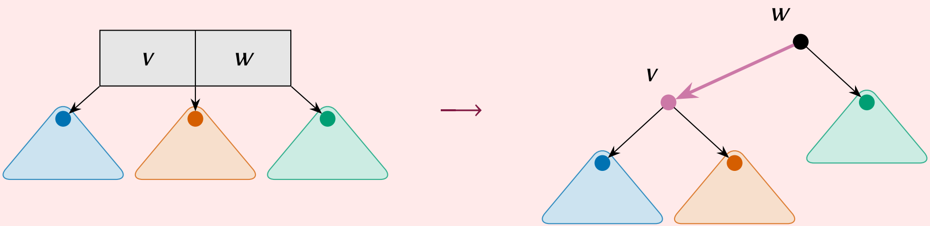
We need some way to identify when a binary search tree structure *represents* a three-node.

From 2-3 trees to *left-leaning* red-black trees

Question: How can we simplify 2-3 trees?

Idea: Turn 2-3 tree nodes into binary search tree structures.

- Two-nodes are already binary search tree nodes.
- Three-nodes can be replaced by a binary search tree structure with two nodes.



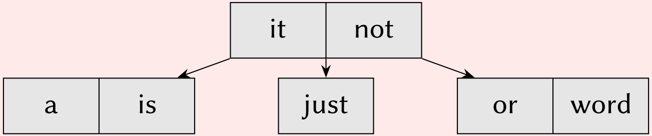
Reusing the addition and removal algorithms from 2-3 trees

We need some way to identify when a binary search tree structure *represents* a three-node.

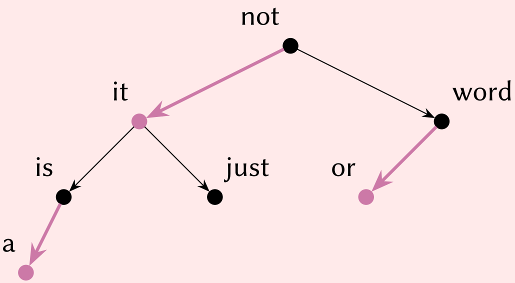
→ Mark the added left-leaning node (with the color red).

From 2-3 trees to *left-leaning* red-black trees

A 2-3 tree

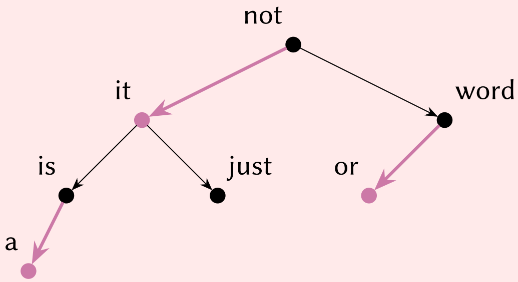


An equivalent *left-leaning* red-black tree



From 2-3 trees to *left-leaning* red-black trees

An equivalent *left-leaning* red-black tree

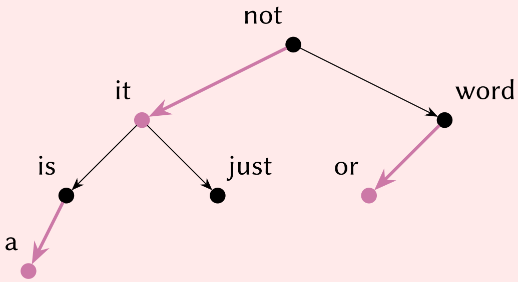


Some usefull properties

1. Every path from root to leaf has at-most $\log_2(N)$ unmarked nodes.
2. Every path from root to leaf has the same number of *unmarked* nodes.
3. No marked nodes “touch” each other.

From 2-3 trees to *left-leaning* red-black trees

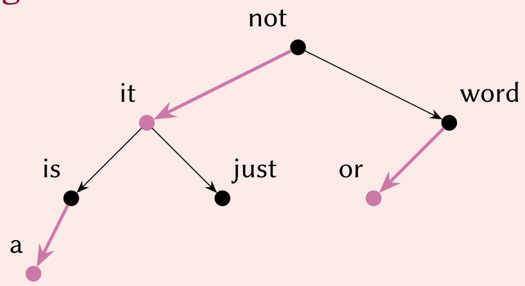
An equivalent *left-leaning* red-black tree



Some usefull properties (that we have to maintain)

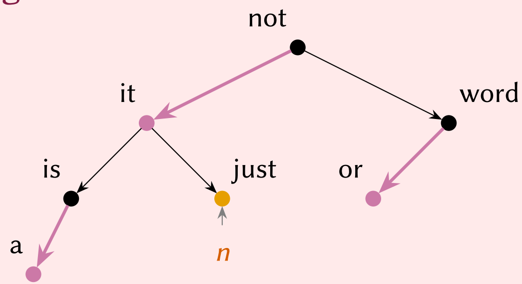
1. Every path from root to leaf has at-most $\log_2(N)$ unmarked nodes.
2. Every path from root to leaf has the same number of *unmarked* nodes.
3. No marked nodes "touch" each other.

Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

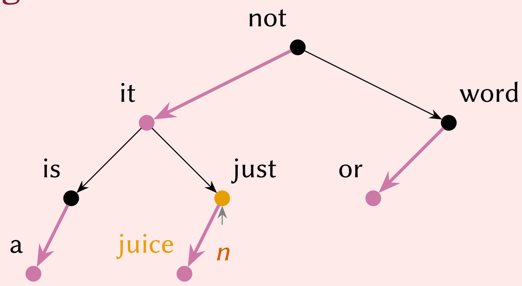
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "juice": we find the leaf *two-node n* holding "just".

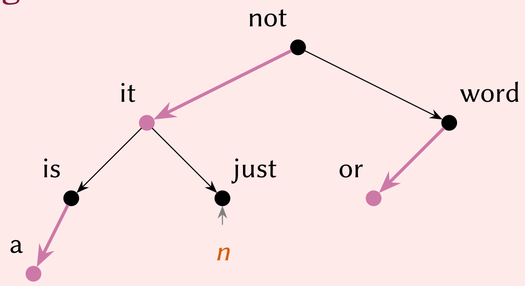
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "juice": we find the leaf *two-node* *n* holding "just".
2. We can turn node *n* into a *three-node*.

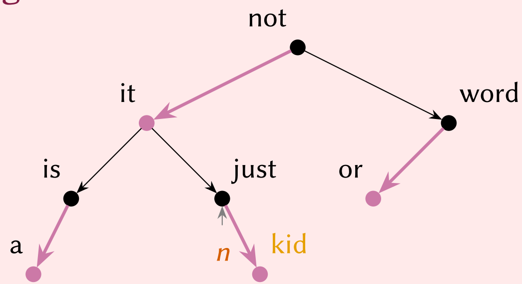
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "kid": we find the leaf *two-node* *n* holding "just".

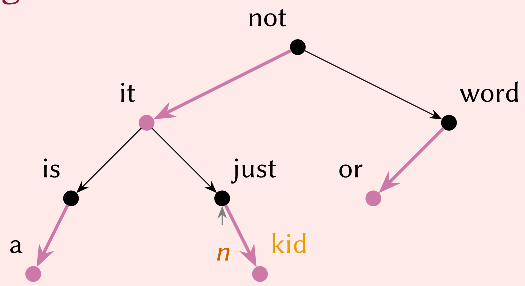
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “kid”: we find the leaf *two-node* *n* holding “just”.
2. We can turn node *n* into a *three-node*, but simply adding “kid” will not do so!

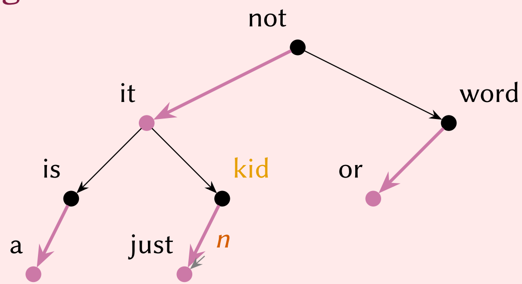
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “kid”: we find the leaf *two-node* *n* holding “just”.
2. We can turn node *n* into a *three-node*, but simply adding “kid” will not do so!
3. We can *rotate left* around *n* to make a proper three-node.

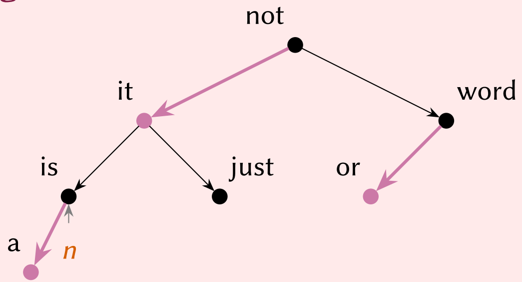
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “kid”: we find the leaf *two-node* *n* holding “just”.
2. We can turn node *n* into a *three-node*, but simply adding “kid” will not do so!
3. We can *rotate left* around *n* to make a proper three-node.

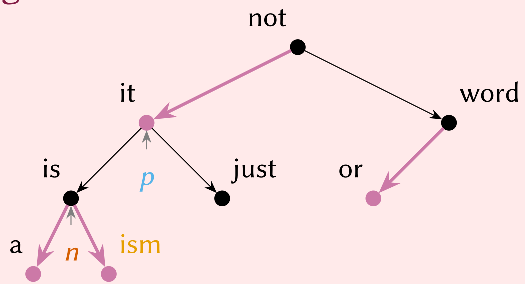
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “ism”: we find the node n holding “is” (part of a *three-node*).

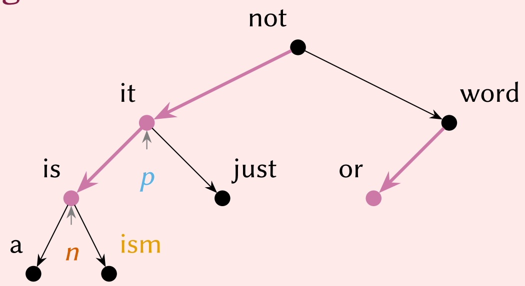
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “ism”: we find the node n holding “is” (part of a *three-node*).
2. We can turn node n into a *three-node*, but simply adding “ism” will not do so!

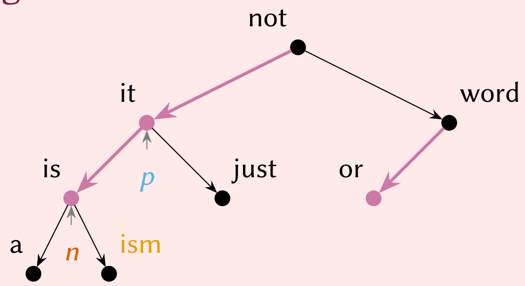
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “ism”: we find the node *n* holding “is” (part of a *three-node*).
2. We can turn node *n* into a *three-node*, but simply adding “ism” will not do so!
3. Push color toward parent *p* of *n*: now marked nodes “touch” each other.

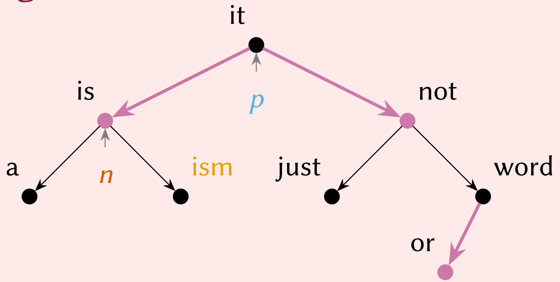
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "ism": we find the node *n* holding "is" (part of a *three-node*).
2. We can turn node *n* into a *three-node*, but simply adding "ism" will not do so!
3. Push color toward parent *p* of *n*: now marked nodes "touch" each other.
4. We can *rotate right* around the parent of *p* toward fixing the marked nodes.

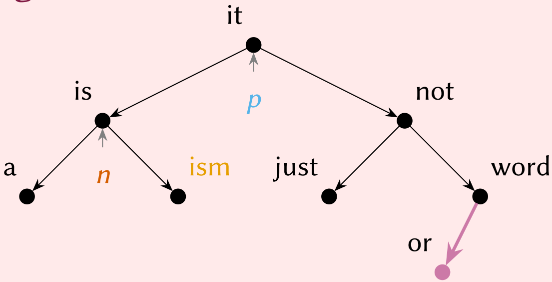
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “ism”: we find the node *n* holding “is” (part of a *three-node*).
2. We can turn node *n* into a *three-node*, but simply adding “ism” will not do so!
3. Push color toward parent *p* of *n*: now marked nodes “touch” each other.
4. We can *rotate right* around the parent of *p* toward fixing the marked nodes.

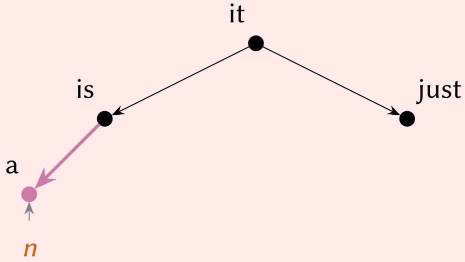
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “ism”: we find the node n holding “is” (part of a *three-node*).
2. We can turn node n into a *three-node*, but simply adding “ism” will not do so!
3. Push color toward parent p of n: now marked nodes “touch” each other.
4. We can *rotate right* around the parent of p toward fixing the marked nodes.
5. Push color toward parent of p (roots stay unmarked).

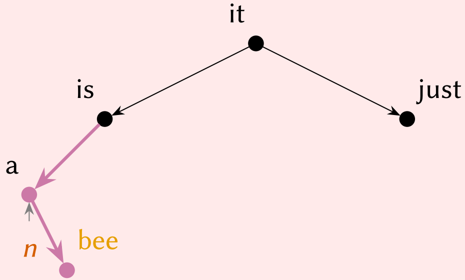
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “bee”: we find the node *n* holding “a” (part of a *three-node*).

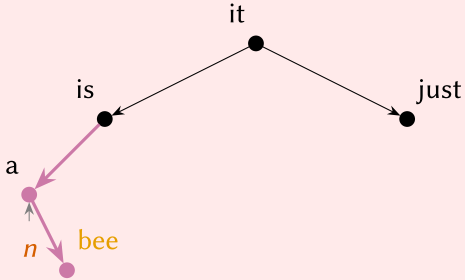
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “bee”: we find the node *n* holding “a” (part of a *three-node*).
2. Simply adding “bee” invalidates the entire structure.

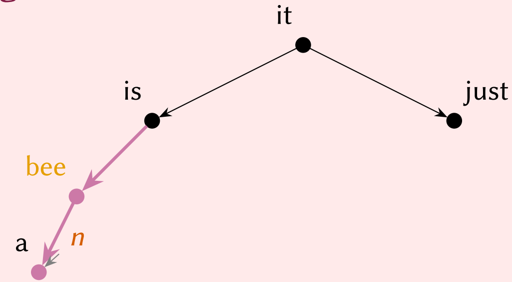
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "bee": we find the node *n* holding "a" (part of a *three-node*).
2. Simply adding "bee" invalidates the entire structure.
3. We can *rotate left* around *n* to turn this case into a previous case!

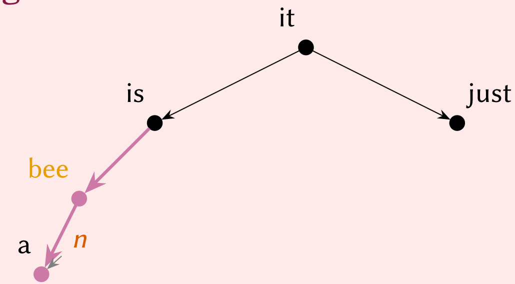
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "bee": we find the node *n* holding "a" (part of a *three-node*).
2. Simply adding "bee" invalidates the entire structure.
3. We can *rotate left* around *n* to turn this case into a previous case!

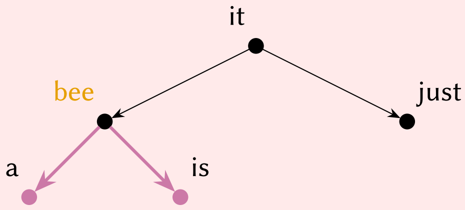
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “bee”: we find the node *n* holding “a” (part of a *three-node*).
2. Simply adding “bee” invalidates the entire structure.
3. We can *rotate left* around *n* to turn this case into a previous case!
4. Marked nodes “touch” each other: we *rotate right* around the node holding “is”.

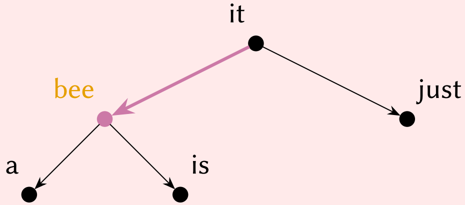
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "bee": we find the node *n* holding "a" (part of a *three-node*).
2. Simply adding "bee" invalidates the entire structure.
3. We can *rotate left* around *n* to turn this case into a previous case!
4. Marked nodes "touch" each other: we *rotate right* around the node holding "is".

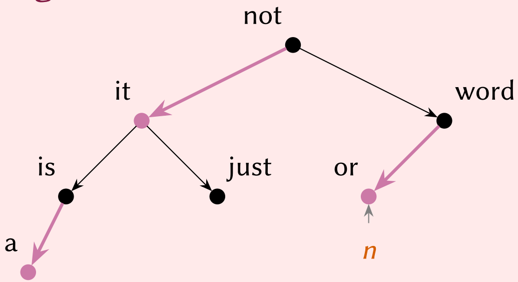
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "bee": we find the node *n* holding "a" (part of a *three-node*).
2. Simply adding "bee" invalidates the entire structure.
3. We can *rotate left* around *n* to turn this case into a previous case!
4. Marked nodes "touch" each other: we *rotate right* around the node holding "is".
5. Push color toward parent.

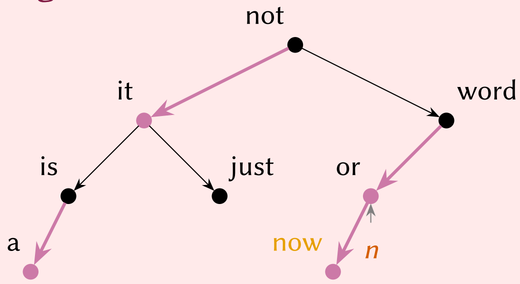
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “nor”: we find the node *n* holding “or” (part of a *three-node*).

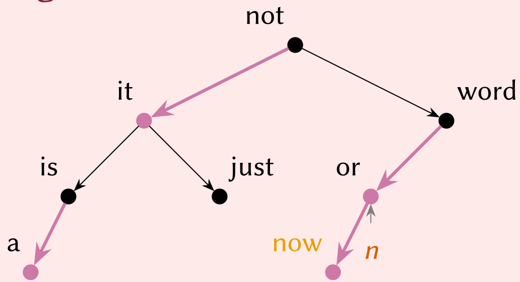
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “nor”: we find the node *n* holding “or” (part of a *three-node*).
2. Simply adding “nor” invalidates the entire structure.

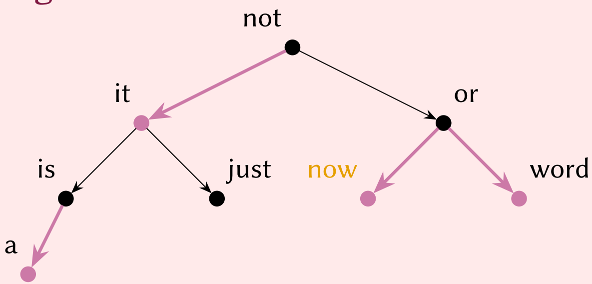
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “nor”: we find the node *n* holding “or” (part of a *three-node*).
2. Simply adding “nor” invalidates the entire structure.
3. This is a previous case: *rotate right*.

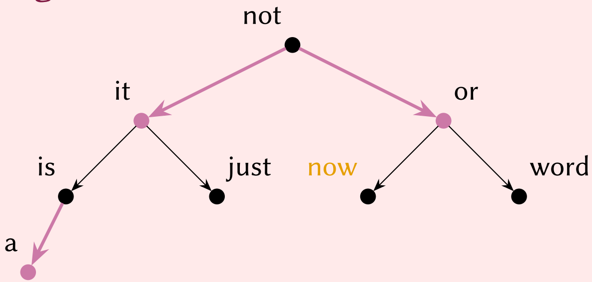
Adding to *left-leaning* red-black trees



Consider adding "juice", "kid", "ism", "bee", and "now"

1. Search for "nor": we find the node *n* holding "or" (part of a *three-node*).
2. Simply adding "nor" invalidates the entire structure.
3. This is a previous case: *rotate right*.

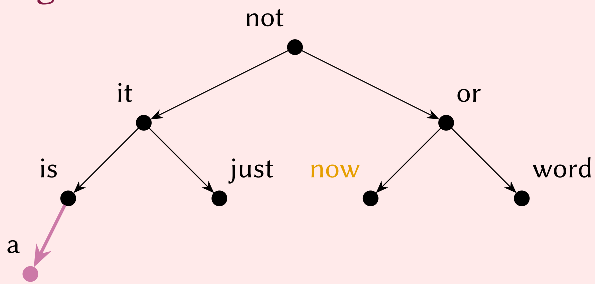
Adding to *left-leaning* red-black trees



Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “nor”: we find the node *n* holding “or” (part of a *three-node*).
2. Simply adding “nor” invalidates the entire structure.
3. This is a previous case: *rotate right*.
4. Push color up.

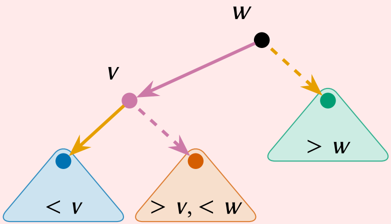
Adding to *left-leaning* red-black trees



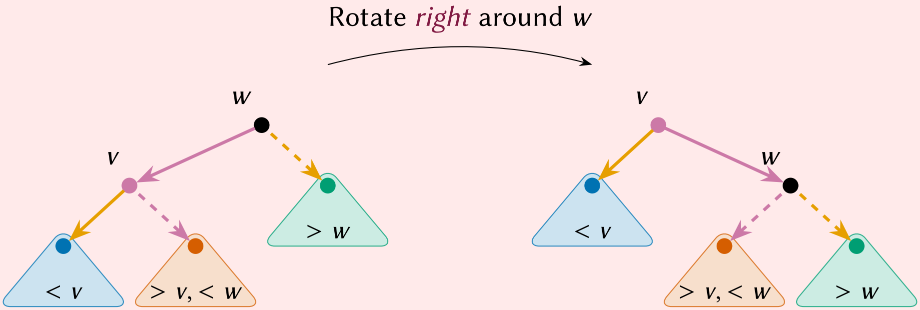
Consider adding “juice”, “kid”, “ism”, “bee”, and “now”

1. Search for “nor”: we find the node *n* holding “or” (part of a *three-node*).
2. Simply adding “nor” invalidates the entire structure.
3. This is a previous case: *rotate right*.
4. Push color up.
5. Push color up (roots stay unmarked).

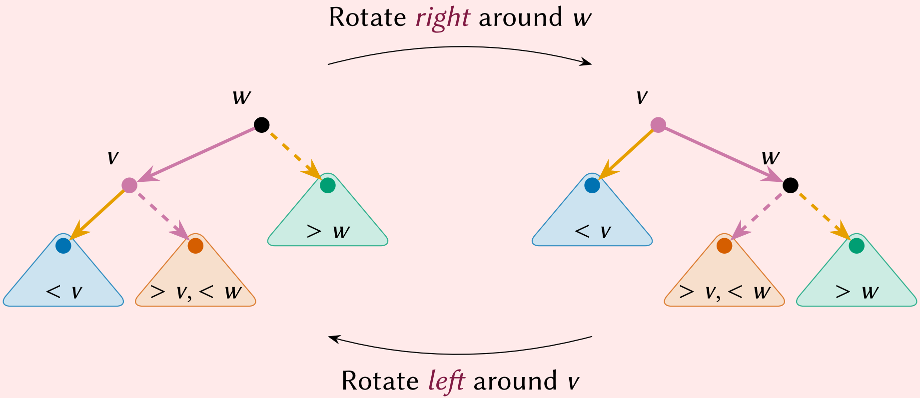
The *rotate left* and *rotate right* operations



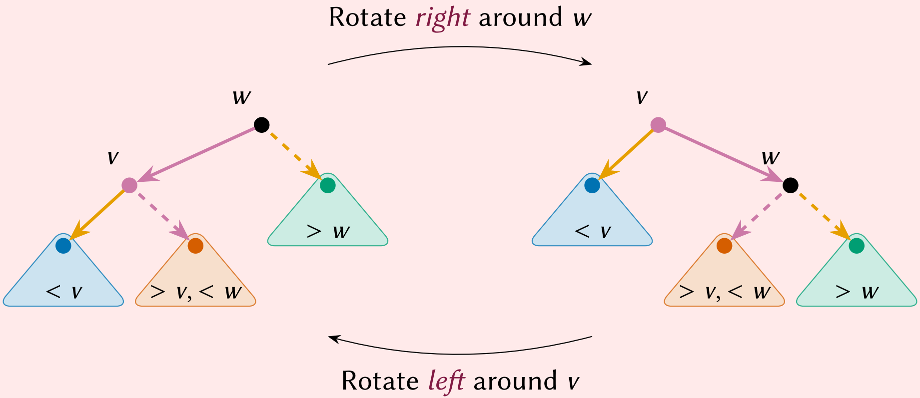
The *rotate left* and *rotate right* operations



The *rotate left* and *rotate right* operations



The *rotate left* and *rotate right* operations

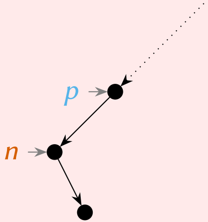


Rotate operations *affect node markings*.

Can be implemented using *only* pointer manipulation.

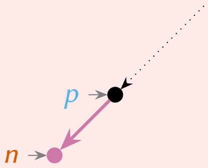
Removing values from *left-leaning* red-black trees (sketch)

Consider a minimum value v at node n with parent p



Removing values from *left-leaning* red-black trees (sketch)

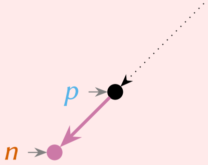
Consider a minimum value v at node n with parent p



- n is marked and has no children.

Removing values from *left-leaning* red-black trees (sketch)

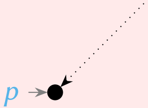
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.

Removing values from *left-leaning* red-black trees (sketch)

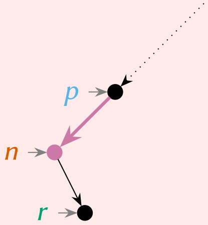
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.

Removing values from *left-leaning* red-black trees (sketch)

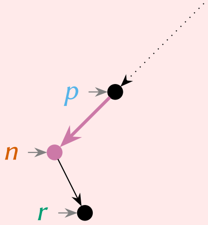
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.
- n is marked and has one (right) child node r.

Removing values from *left-leaning* red-black trees (sketch)

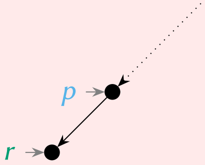
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.
- n is marked and has one (right) child node r.
  Simple: Replace n by r, which has zero consequences on the structure of the tree.

Removing values from *left-leaning* red-black trees (sketch)

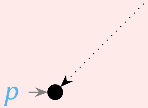
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.
- n is marked and has one (right) child node r.
  Simple: Replace n by r, which has zero consequences on the structure of the tree.

Removing values from *left-leaning* red-black trees (sketch)

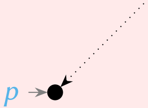
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.
- n is marked and has one (right) child node r.
  Simple: Replace n by r, which has zero consequences on the structure of the tree.
- n is *not* marked.

Removing values from *left-leaning* red-black trees (sketch)

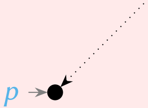
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.
- n is marked and has one (right) child node r.
  Simple: Replace n by r, which has zero consequences on the structure of the tree.
- n is *not* marked $\longrightarrow$ Complex: Removing n invalidates the structure of the tree.

Removing values from *left-leaning* red-black trees (sketch)

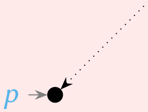
Consider a minimum value v at node n with parent p



- n is marked and has no children.
  Simple: Removing has zero consequences on the structure of the tree.
- n is marked and has one (right) child node r.
  Simple: Replace n by r, which has zero consequences on the structure of the tree.
- n is *not* marked $\longrightarrow$ Complex: Removing n invalidates the structure of the tree.
  Idea: Ensure that n is marked.

Removing values from *left-leaning* red-black trees (sketch)

Consider a minimum value v at node n with parent p

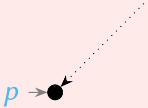


Idea: Ensure that n is marked.

- We can *introduce* marked nodes at the root of the tree.
- We can push marked nodes down the tree using *rotates* toward the minimum value.

Removing values from *left-leaning* red-black trees (sketch)

Consider a minimum value v at node n with parent p



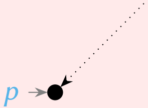
Idea: Ensure that n is marked.

- We can *introduce* marked nodes at the root of the tree.
- We can push marked nodes down the tree using *rotates* toward the minimum value.

We have seen the reverse while *adding* values.

Removing values from *left-leaning* red-black trees (sketch)

Consider a minimum value v at node n with parent p

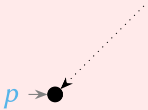


Generalization: Remove arbitrary values.

- Replace arbitrary values by their successor.
- Removing successor: generalize the methods to remove the minimum from a tree.

Removing values from *left-leaning* red-black trees (sketch)

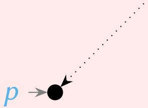
Consider a minimum value v at node n with parent p



Removal is possible with only local tree modifications along the path from root to value.

Removing values from *left-leaning* red-black trees (sketch)

Consider a minimum value v at node n with parent p



Removal is possible with only local tree modifications along the path from root to value.

Many minute details to deal with in a plethora of cases.

Conclusion: Left-leaning red-black trees

Some usefull properties (that we can maintain)

1. Every path from root to leaf has at-most $\log_2(N)$ unmarked nodes.
2. Every path from root to leaf has the same number of *unmarked* nodes.
3. No marked nodes "touch" each other.

Conclusion: Left-leaning red-black trees

Some usefull properties (that we can maintain)

1. Every path from root to leaf has at-most $\log_2(N)$ unmarked nodes.
2. Every path from root to leaf has the same number of *unmarked* nodes.
3. No marked nodes "touch" each other.

Paths from root to leafs have length *at-most* $2\log_2(N)$:
all operations of interest in worst-case $\Theta\left(\log_2(N)\right)$.

Final notes on binary search trees

We looked at *left-leaning* red-black trees.
In practice, one typically uses ordinary red-back trees:
Very similar, just *more cases* to consider when adding or removing values.

Final notes on binary search trees

We looked at *left-leaning* red-black trees.
In practice, one typically uses ordinary red-back trees:
Very similar, just *more cases* to consider when adding or removing values.

	C++	Java
Set	`std::set`	`java.util.TreeSet`
Dictionary	`std::map`	`java.util.TreeMap`
Set (duplicates)	`std::multiset`	
Dictionary (duplicates)	`std::multimap`	

Final notes on binary search trees

We looked at *left-leaning* red-black trees.
In practice, one typically uses ordinary red-back trees:
Very similar, just *more cases* to consider when adding or removing values.

	C++	Java
Set	`std::set`	`java.util.TreeSet`
Dictionary	`std::map`	`java.util.TreeMap`
Set (duplicates)	`std::multiset`	
Dictionary (duplicates)	`std::multimap`	

Variants of search trees are used *everywhere*: file systems, database systems, ...

Faster sets and dictionaries: beyond $\log_2(N)$

Consider the following variant of WordCount

Algorithm GradeCount(*stream*):

Input: *stream* is a sequence of grades, each in $0, \ldots, 10$.

1: $grades := [0 \mid 0 \leq i \leq 10]$.
2: **for all** grade g from *stream* **do**
3: $\quad grades[g] := grades[g] + 1$.
4: output each pair $(i \mapsto grades[i])$, $0 \leq i \leq 10$.

Result: output a histogram of the grades in *stream*.

Faster sets and dictionaries: beyond $\log_2(N)$

Consider the following variant of WordCount

Algorithm GradeCount(*stream*):

Input: *stream* is a sequence of grades, each in $0, \ldots, 10$.

1: $\mathit{grades} := [0 \mid 0 \le i \le 10]$.
2: **for all** grade g from *stream* **do**
3: $\quad\mathit{grades}[g] := \mathit{grades}[g] + 1$.
4: output each pair $(i \mapsto \mathit{grades}[i])$, $0 \le i \le 10$.

Result: output a histogram of the grades in *stream*.

grades is an *array* that essentially serves as a *dictionary*
in which *grades* are keys and a grade-count is the associated value.

Faster sets and dictionaries: beyond $\log_2(N)$

Consider the following variant of WordCount

Algorithm GradeCount(*stream*):

Input: *stream* is a sequence of grades, each in $0, \dots, 10$.

1: $grades := [0 \mid 0 \leq i \leq 10]$.
2: **for all** grade g from *stream* **do**
3: $\quad grades[g] := grades[g] + 1$.
4: output each pair $(i \mapsto grades[i])$, $0 \leq i \leq 10$.

Result: output a histogram of the grades in *stream*.

grades is an *array* that essentially serves as a *dictionary*
in which *grades* are keys and a grade-count is the associated value.

Worst-case complexity only $\Theta(|stream|)$.

Toward using arrays as dictionaries

An array $L[0 \ldots N)$ maps *positions* $0, \ldots, N$ onto values.
For sets: the value could be the key itself.

Toward using arrays as dictionaries

An array $L[0\ldots N)$ maps *positions* $0, \ldots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings “a”, “word”, “is”, “just”, “or”, “it”, “not”.

Toward using arrays as dictionaries

An array $L[0 \ldots N)$ maps *positions* $0, \ldots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings “a”, “word”, “is”, “just”, “or”, “it”, “not”.

Generalizing array-dictionaries

Given an arbitrary set of *keys* $\mathcal{K}$, we need a function $h : \mathcal{K} \to \{0, \ldots, N-1\}$ that maps these keys to array positions.

Toward using arrays as dictionaries

An array $L[0 \ldots N)$ maps *positions* $0, \ldots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings "a", "word", "is", "just", "or", "it", "not".

Generalizing array-dictionaries

Given an arbitrary set of *keys* $\mathcal{K}$, we need a function $h : \mathcal{K} \to \{0, \ldots, N-1\}$
that maps these keys to array positions $\to$ a *hash function*.

Toward using arrays as dictionaries

$L[0 \ldots 10)$:

0:	
1:	
2:	
3:	
4:	
5:	
6:	
7:	
8:	
9:	

Toward using arrays as dictionaries

Consider h : Strings $\to \{0, \dots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

$L[0 \dots 10)$:

0:	
1:	
2:	
3:	
4:	
5:	
6:	
7:	
8:	
9:	

Toward using arrays as dictionaries

Consider h : Strings $\rightarrow \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	
"word"	
"is"	
"just"	
"or"	
"it"	
"not"	

$L[0 \ldots 10)$:

0:	
1:	
2:	
3:	
4:	
5:	
6:	
7:	
8:	
9:	

Toward using arrays as dictionaries

Consider h : Strings $\to \{0, \dots 9\}$ with

First character	$h(v)$
‘a’, ‘k’, ‘u’	0
‘b’, ‘l’, ‘v’	1
‘c’, ‘m’, ‘w’	2
‘d’, ‘n’, ‘x’	3
‘e’, ‘o’, ‘y’	4
‘f’, ‘p’, ‘z’	5
‘g’, ‘q’	6
‘h’, ‘r’	7
‘i’, ‘s’	8
‘j’, ‘t’	9

w	$h(w)$
“a”	0
“word”	
“is”	
“just”	
“or”	
“it”	
“not”	

$L[0 \dots 10)$:

0:	a
1:	
2:	
3:	
4:	
5:	
6:	
7:	
8:	
9:	

Toward using arrays as dictionaries

Consider h : Strings $\rightarrow \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	0
"word"	2
"is"	
"just"	
"or"	
"it"	
"not"	

$L[0 \ldots 10)$:

0:	a
1:	
2:	word
3:	
4:	
5:	
6:	
7:	
8:	
9:	

Toward using arrays as dictionaries

Consider h : Strings $\to \{0, \dots 9\}$ with

First character	$h(v)$
‘a’, ‘k’, ‘u’	0
‘b’, ‘l’, ‘v’	1
‘c’, ‘m’, ‘w’	2
‘d’, ‘n’, ‘x’	3
‘e’, ‘o’, ‘y’	4
‘f’, ‘p’, ‘z’	5
‘g’, ‘q’	6
‘h’, ‘r’	7
‘i’, ‘s’	8
‘j’, ‘t’	9

w	$h(\mathrm{w})$
“a”	0
“word”	2
“is”	8
“just”	
“or”	
“it”	
“not”	

$L[0 \dots 10)$:

0:	a
1:	
2:	word
3:	
4:	
5:	
6:	
7:	
8:	is
9:	

Toward using arrays as dictionaries

Consider h : Strings $\to \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	0
"word"	2
"is"	8
"just"	9
"or"	
"it"	
"not"	

$L[0 \ldots 10)$:

0:	a
1:	
2:	word
3:	
4:	
5:	
6:	
7:	
8:	is
9:	just

Toward using arrays as dictionaries

Consider h : Strings $\rightarrow \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	0
"word"	2
"is"	8
"just"	9
"or"	4
"it"	
"not"	

$L[0 \ldots 10)$:

0:	a
1:	
2:	word
3:	
4:	or
5:	
6:	
7:	
8:	is
9:	just

Toward using arrays as dictionaries

Consider h : Strings $\to \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	0
"word"	2
"is"	8
"just"	9
"or"	4
"it"	8
"not"	

$L[0 \ldots 10)$:

0:	a
1:	
2:	word
3:	
4:	or
5:	
6:	
7:	
8:	is
9:	just

it?

Toward using arrays as dictionaries

Consider h : Strings $\rightarrow \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	0
"word"	2
"is"	8
"just"	9
"or"	4
"it"	8
"not"	

$L[0 \ldots 10)$:

0:	a	
1:		
2:	word	
3:		
4:	or	
5:		
6:		
7:		
8:	is	it? $\leftarrow$ a *collision*!
9:	just	

Toward using arrays as dictionaries

Consider h : Strings $\rightarrow \{0, \ldots 9\}$ with

First character	$h(v)$
'a', 'k', 'u'	0
'b', 'l', 'v'	1
'c', 'm', 'w'	2
'd', 'n', 'x'	3
'e', 'o', 'y'	4
'f', 'p', 'z'	5
'g', 'q'	6
'h', 'r'	7
'i', 's'	8
'j', 't'	9

w	$h(w)$
"a"	0
"word"	2
"is"	8
"just"	9
"or"	4
"it"	8
"not"	3

$L[0 \ldots 10)$:

0:	a
1:	
2:	word
3:	not
4:	or
5:	
6:	
7:	
8:	is
9:	just

Toward using arrays as dictionaries

An array $L[0 \dots N)$ maps *positions* $0, \dots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings “a”, “word”, “is”, “just”, “or”, “it”, “not”.

Generalizing array-dictionaries

Given an arbitrary set of *keys* $\mathcal{K}$, we need a function $h : \mathcal{K} \to \{0, \dots, N-1\}$
that maps these keys to array positions $\to$ a *hash function*.

We want to *prevent* collisions

Toward using arrays as dictionaries

An array $L[0 \dots N)$ maps *positions* $0, \dots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings “a”, “word”, “is”, “just”, “or”, “it”, “not”.

Generalizing array-dictionaries

Given an arbitrary set of *keys* $\mathcal{K}$, we need a function $h : \mathcal{K} \to \{0, \dots, N-1\}$ that maps these keys to array positions $\longrightarrow$ a *hash function*.

We want to *prevent* collisions

- What if $|\mathcal{K}|$ is very large?
  For example, the number of strings is infinite.
- What if N is very small?
  For example, to save memory when we only aim to store a few keys.

Toward using arrays as dictionaries

An array $L[0 \ldots N)$ maps *positions* $0, \ldots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings “a”, “word”, “is”, “just”, “or”, “it”, “not”.

Generalizing array-dictionaries

Given an arbitrary set of *keys* $\mathcal{K}$, we need a function $h : \mathcal{K} \to \{0, \ldots, N - 1\}$
that maps these keys to array positions $\longrightarrow$ a *hash function*.

We want to *prevent* collisions

- What if $|\mathcal{K}|$ is very large?
  For example, the number of strings is infinite.
- What if N is very small?
  For example, to save memory when we only aim to store a few keys.

We also want “*cheap*” hash functions to maximize performance.

Toward using arrays as dictionaries

An array $L[0 \ldots N)$ maps *positions* $0, \ldots, N$ onto values.
For sets: the value could be the key itself.

Very resitrictive: most *keys* are not integers in a very small range.
For example, keys could be strings “a”, “word”, “is”, “just”, “or”, “it”, “not”.

Generalizing array-dictionaries

Given an arbitrary set of *keys* $\mathcal{K}$, we need a function $h : \mathcal{K} \to \{0, \ldots, N-1\}$
that maps these keys to array positions $\longrightarrow$ a *hash function*.

We have to *deal* with collisions

- What if $|\mathcal{K}|$ is very large?
  For example, the number of strings is infinite.
- What if N is very small?
  For example, to save memory when we only aim to store a few keys.

We also want “*cheap*” hash functions to maximize performance.

Hash tables

A *hash table* is a data structure that uses a *hash function* that maps *values* to array positions that can *hold that value*.

Hash tables

A *hash table* is a data structure that uses a *hash function*
that maps *values* to array positions that can *hold that value*.

The way a hash table *holds values* is determined by how the table deals with *collisions*:
Typically determines the design of the data structure.

Hash tables

A *hash table* is a data structure that uses a *hash function* that maps *values* to array positions that can *hold that value*.

The way a hash table *holds values* is determined by how the table deals with *collisions*: Typically determines the design of the data structure.

We will look at two main flavors of hash tables:

Chaining Use a linked list to store *collisions*.

Linear probing Store *collisions* consecutively in the array.

The *uniform hashing* assumption

Let $h : \mathcal{K} \to \{0, \ldots, N-1\}$ be a hash function.
We *assume* that the hash function distributes the values in $\mathcal{K}$ *uniformly and independently* among the positions $\{0, \ldots, N-1\}$.

The *uniform hashing* assumption

Let $h : \mathcal{K} \to \{0, \ldots, N-1\}$ be a hash function.
We *assume* that the hash function distributes the values in $\mathcal{K}$
uniformly and independently among the positions $\{0, \ldots, N-1\}$.

For any two distinct values $v_1, v_2 \in \mathcal{K}$, we have $h(v_1) = h(v_2)$ with a probability of $\frac{1}{N}$.

The *uniform hashing* assumption

Let $h : \mathcal{K} \to \{0, \ldots, N-1\}$ be a hash function.
We *assume* that the hash function distributes the values in $\mathcal{K}$ *uniformly and independently* among the positions $\{0, \ldots, N-1\}$.

For any two distinct values $v_1, v_2 \in \mathcal{K}$, we have $h(v_1) = h(v_2)$ with a probability of $\frac{1}{N}$.

Using this assumption, we can analyze the *expected behavior* of hash tables.

The *uniform hashing* assumption

Let $h : \mathcal{K} \to \{0, \ldots, N-1\}$ be a hash function.
We *assume* that the hash function distributes the values in $\mathcal{K}$ *uniformly and independently* among the positions $\{0, \ldots, N-1\}$.

For any two distinct values $v_1, v_2 \in \mathcal{K}$, we have $h(v_1) = h(v_2)$ with a probability of $\frac{1}{N}$.

Using this assumption, we can analyze the *expected behavior* of hash tables.

Some settings allow a collision-free hash function: *perfect hashing*.
For example: the hash function $h(i) = i$ we used in GradeCount.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Contains value v Look up the linked list S at $L[h(v)]$,
search v in S (e.g., using a LinearSearch variant).

Adding value v Look up the linked list S at $L[h(v)]$,
add v to S if $v \notin S$ (sets do not have duplicates).

Removing value v Look up the linked list S at $L[h(v)]$,
remove v from S if $v \in S$.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\to \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	h(w)
“a”	
“word”	
“is”	
“just”	
“or”	
“it”	
“not”	

$L[0 \ldots 7)$:

0:	`@null`
1:	`@null`
2:	`@null`
3:	`@null`
4:	`@null`
5:	`@null`
6:	`@null`

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	
“is”	
“just”	
“or”	
“it”	
“not”	

$L[0 \ldots 7)$:

0:	@123A
1:	@null
2:	@null
3:	@null
4:	@null
5:	@null
6:	@null

@123A:

item: “a”
next: @null

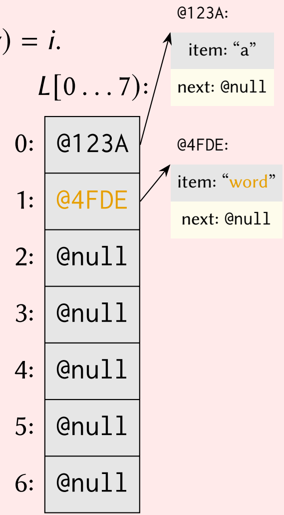
Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“is”	
“just”	
“or”	
“it”	
“not”	



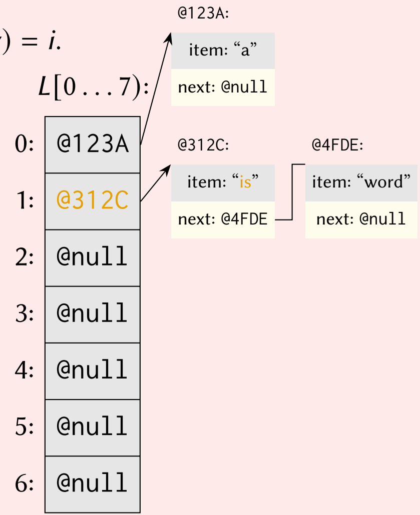
Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \dots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“is”	1
“just”	
“or”	
“it”	
“not”	



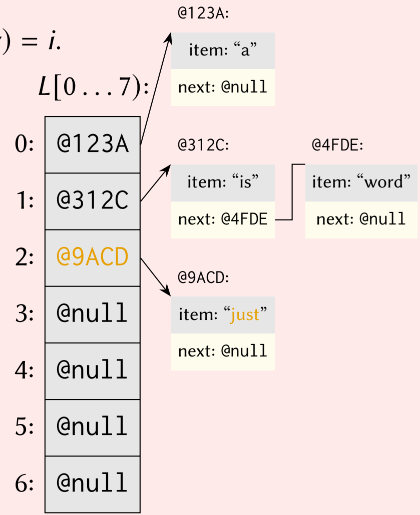
Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“is”	1
“just”	2
“or”	
“it”	
“not”	



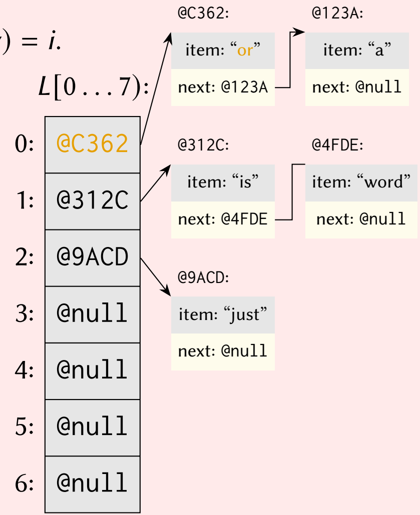
Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“is”	1
“just”	2
“or”	0
“it”	
“not”	



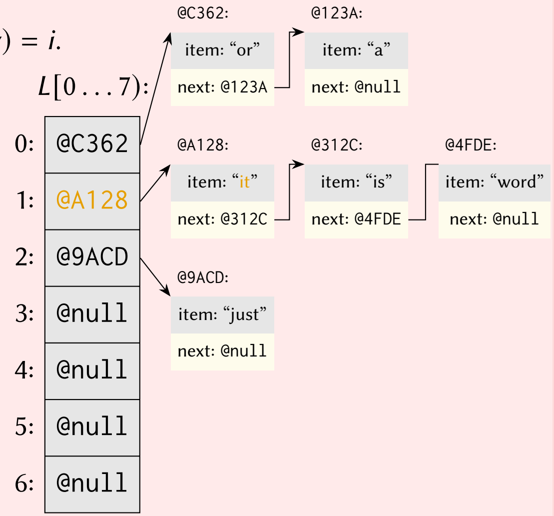
Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“is”	1
“just”	2
“or”	0
“it”	1
“not”	



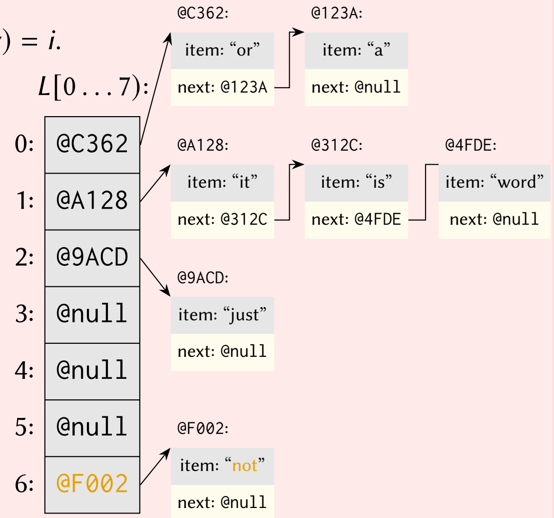
Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

h : Strings $\longrightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“is”	1
“just”	2
“or”	0
“it”	1
“not”	6



Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Analysis

Consider a hash table with N positions, holding M values.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Analysis

Consider a hash table with N positions, holding M values.

- On average, each linked list holds $\frac{M}{N}$ values.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Analysis

Consider a hash table with N positions, holding M values.

- On average, each linked list holds $\frac{M}{N}$ values.
- *If* the uniform hashing assumption holds,
  then adding or removing random values will cost an expected $\Theta\left(1 + \frac{M}{N}\right)$.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Analysis

Consider a hash table with N positions, holding M values.

- On average, each linked list holds $\frac{M}{N}$ values.
- *If* the uniform hashing assumption holds,
  then adding or removing random values will cost an expected $\Theta\left(1 + \frac{M}{N}\right)$.
- Worst-case: $\Theta(N)$ (all values end up in a single linked list).

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Analysis

Consider a hash table with N positions, holding M values.

- On average, each linked list holds $\frac{M}{N}$ values.
- *If* the uniform hashing assumption holds,
  then adding or removing random values will cost an expected $\Theta\left(1 + \frac{M}{N}\right)$.
- Worst-case: $\Theta(N)$ (all values end up in a single linked list).

- For somewhat decent hash functions and $N > M$,
  adding and removing values are $\Theta(1)$ in practice.

Hashing with chaining

Idea: the hash table is an array of linked lists,
the i-th linked list holding all values v with $h(v) = i$.

Analysis

Consider a hash table with N positions, holding M values.

- On average, each linked list holds $\frac{M}{N}$ values.
- *If* the uniform hashing assumption holds,
  then adding or removing random values will cost an expected $\Theta\left(1 + \frac{M}{N}\right)$.
- Worst-case: $\Theta(N)$ (all values end up in a single linked list).

- For somewhat decent hash functions and $N > M$,
  adding and removing values are $\Theta(1)$ in practice.
- Bad hash functions exist.
  For example, the hash function we used in our examples.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Contains value v	Inspect each consecutive non-free position j starting at $h(v)$, return if $L[j] = v$ holds for any such position.
Adding value v	Look up the first free position $j \geq h(v)$ in L, set $L[j] := v$ if we did not find v in any of the inspected positions.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Contains value v Inspect each consecutive non-free position j starting at $h(v)$, return if $L[j] = v$ holds for any such position.

Adding value v Look up the first free position $j \geq h(v)$ in L, set $L[j] := v$ if we did not find v in any of the inspected positions.

How to remove a value?

Removing values breaks consecutive sequences of non-free positions!

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

h : Strings $\rightarrow \{0, \dots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	h(w)
“a”	
“word”	
“just”	
“is”	
“or”	
“not”	
“now”	

$L[0 \dots 7)$:

0:	
1:	
2:	
3:	
4:	
5:	
6:	

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \rightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	
“just”	
“is”	
“or”	
“not”	
“now”	

$L[0 \ldots 7]$:

0:	“a”
1:	
2:	
3:	
4:	
5:	
6:	

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

h : Strings $\rightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“just”	
“is”	
“or”	
“not”	
“now”	

$L[0 \ldots 7)$:

0:	“a”
1:	“word”
2:	
3:	
4:	
5:	
6:	

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \rightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“just”	2
“is”	
“or”	
“not”	
“now”	

$L[0 \ldots 7)$:

0:	“a”
1:	“word”
2:	“just”
3:	
4:	
5:	
6:	

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \rightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“just”	2
“is”	1
“or”	
“not”	
“now”	

$L[0 \ldots 7]$:

0:	“a”
1:	“word”
2:	“just”
3:	“is”
4:	
5:	
6:	

Occupied!

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \to \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“just”	2
“is”	1
“or”	0
“not”	
“now”	

$L[0 \ldots 7)$:

0:	“a”	Occupied!
1:	“word”	Occupied!
2:	“just”	Occupied!
3:	“is”	Occupied!
4:	“or”	
5:		
6:		

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \to \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“just”	2
“is”	1
“or”	0
“not”	6
“now”	

$L[0 \ldots 7)$:

0:	“a”
1:	“word”
2:	“just”
3:	“is”
4:	“or”
5:	
6:	“not”

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

h : Strings $\rightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

w	$h(w)$
“a”	0
“word”	1
“just”	2
“is”	1
“or”	0
“not”	6
“now”	6

$L[0 \ldots 7)$:

0:	“a”	Occupied! (*wrap around*)
1:	“word”	
2:	“just”	
3:	“is”	
4:	“or”	
5:	“nor”	
6:	“not”	Occupied!

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \to \{0, \dots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

Consider removing “word”, by simply erasing the value.

$L[0 \dots 7)$:

0:	“a”
1:	“word”
2:	“just”
3:	“is”
4:	“or”
5:	“nor”
6:	“not”

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \rightarrow \{0, \ldots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

Consider removing “word”, by simply erasing the value.

$L[0 \ldots 7)$:

0:	“a”
1:	
2:	“just”
3:	“is”
4:	“or”
5:	“nor”
6:	“not”

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \to \{0, \dots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

Consider removing “word”, by simply erasing the value.

How can we find “just”, “is”, “or”, “nor”?

$L[0 \dots 7)$:

0:	“a”
1:	
2:	“just”
3:	“is”
4:	“or”
5:	“nor”
6:	“not”

At wrong positions!

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Contains value v Inspect each consecutive non-free position j starting at $h(v)$, return if $L[j] = v$ holds for any such position.

Adding value v Look up the first free position $j \geq h(v)$ in L, set $L[j] := v$ if we did not find v in any of the inspected positions.

How to remove a value at position j?

Removing values breaks consecutive sequences of non-free positions!

Option 1 reinsert all values in non-free positions following position j.

Option 2 set $L[j] := \textsc{Removed}$ with $\textsc{Removed}$ a special-purpose value. When searching: $\textsc{Removed}$ is unequal to any value.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Contains value v Inspect each consecutive non-free position j starting at $h(v)$, return if $L[j] = v$ holds for any such position.

Adding value v Look up the first free position $j \geq h(v)$ in L, set $L[j] := v$ if we did not find v in any of the inspected positions.

How to remove a value at position j?

Removing values breaks consecutive sequences of non-free positions!

Option 1 reinsert all values in non-free positions following position j.

Option 2 set $L[j] :=$ REMOVED with REMOVED a special-purpose value. When searching: REMOVED is unequal to any value.

Option 1 is costlier during removal, but cheaper afterwards.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

$h : \text{Strings} \to \{0, \dots 6\}$

First character	$h(v)$
‘a’, ‘h’, ‘o’, ‘v’	0
“b’, ‘i’, ‘p’, ‘w’	1
“c’, ‘j’, ‘q’, ‘x’	2
“d’, ‘k’, ‘r’, ‘y’	3
“e’, ‘l’, ‘s’, ‘z’	4
“f’, ‘m’, ‘t’	5
“g’, ‘n’, ‘u’	6

Consider removing “word”,
by simply erasing the value.

How can we find
“just”, “is”, “or”, “nor”?

Option 1.
We reinsert these three values.

$L[0 \dots 7)$:

0:	“a”
1:	
2:	“just”
3:	“is”
4:	“or”
5:	“nor”
6:	“not”

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

h : Strings $\to \{0, \dots 6\}$

First character	$h(v)$
'a', 'h', 'o', 'v'	0
"b', 'i', 'p', 'w'	1
"c', 'j', 'q', 'x'	2
"d', 'k', 'r', 'y'	3
"e', 'l', 's', 'z'	4
"f', 'm', 't'	5
"g', 'n', 'u'	6

Consider removing "word",
by simply erasing the value.

How can we find
"just", "is", "or", "nor"?

Option 1.
We reinsert these three values.

$L[0 \dots 7)$:

0:	"a"
1:	"is"
2:	"just"
3:	"or"
4:	"nor"
5:	
6:	"not"

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

- *If* the uniform hashing assumption holds,
  then the i-th position holds a value with probability α.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

- *If* the uniform hashing assumption holds,
  then the i-th position holds a value with probability α
  and the probability that j consecutive positions hold a value is at-most α^j.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

- *If* the uniform hashing assumption holds,
  then the i-th position holds a value with probability α
  and the probability that j consecutive positions hold a value is at-most α^j.
- To find a non-existing value (*adding*), we expect to inspect at-most

$$1 + \alpha + \alpha^2 + \alpha^3 + \cdots + \alpha^N$$

  positions.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

- *If* the uniform hashing assumption holds,
  then the i-th position holds a value with probability α
  and the probability that j consecutive positions hold a value is at-most α^j.
- To find a non-existing value (*adding*), we expect to inspect at-most

$$1 + \alpha + \alpha^2 + \alpha^3 + \cdots + \alpha^N \leq \sum_{i=0}^{\infty} \alpha^i$$

  positions.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

- *If* the uniform hashing assumption holds,
  then the i-th position holds a value with probability α
  and the probability that j consecutive positions hold a value is at-most α^j.
- To find a non-existing value (*adding*), we expect to inspect at-most

$$1 + \alpha + \alpha^2 + \alpha^3 + \cdots + \alpha^N \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

  positions.

Hashing with linear probing

Idea: the hash table holds all values directly,
the value v will be stored at the first free position at-or-after $h(v) = i$.
At-or-after with *wrap around*: position 0 comes right after the last position.

Analysis

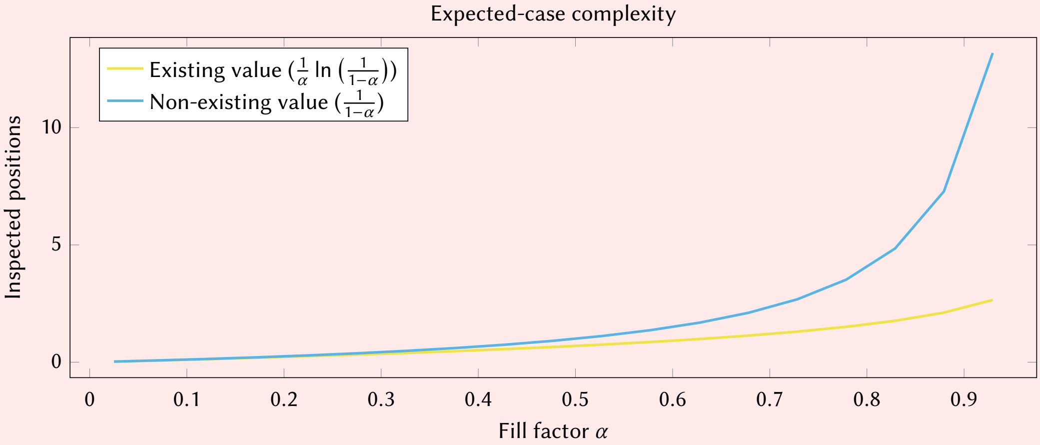
Consider a hash table with N positions, holding M values. Let $\alpha = \frac{M}{N}$ be the *fill factor*.

- *If* the uniform hashing assumption holds,
  then the i-th position holds a value with probability α
  and the probability that j consecutive positions hold a value is at-most α^j.
- To find a non-existing value (*adding*), we expect to inspect at-most

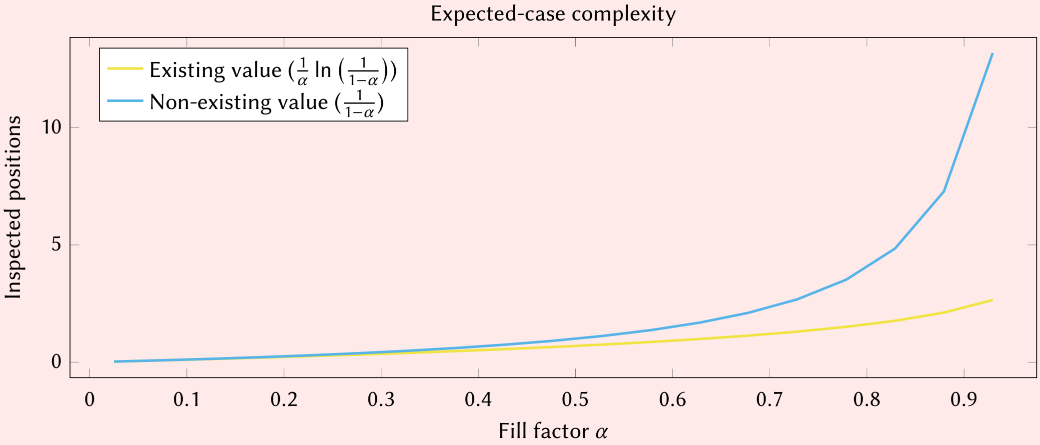
$$1 + \alpha + \alpha^2 + \alpha^3 + \cdots + \alpha^N \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$

  positions.
- To find an existing value (*removing*), we expect to inspect at-most $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ positions.

Hashing with linear probing



Hashing with linear probing



For somewhat decent hash functions and $N \ggg M$,
adding and removing values are $\Theta(1)$ in practice.

Hash tables and functions in practice

Hash tables provide a balance between memory usage and runtime cost:

- With mostly-empty tables (high memory usage), collisions are expected to be rare (low runtime cost).
- With mostly-full tables (low memory usage), collisions are expected to be frequent (high runtime cost).

Hash tables and functions in practice

Hash tables provide a balance between memory usage and runtime cost:

- With mostly-empty tables (high memory usage),
  collisions are expected to be rare (low runtime cost).
- With mostly-full tables (low memory usage),
  collisions are expected to be frequent (high runtime cost).

In practice, one typically *resizes* the hash table when it gets too full.

Hash tables and functions in practice

Hash tables provide a balance between memory usage and runtime cost:

- With mostly-empty tables (high memory usage),
  collisions are expected to be rare (low runtime cost).
- With mostly-full tables (low memory usage),
  collisions are expected to be frequent (high runtime cost).

In practice, one typically *resizes* the hash table when it gets too full.

This requires a *family of hash functions* $h_N : \mathcal{K} \to \{0, \dots, N-1\}$.

Hash tables and functions in practice

Hash tables provide a balance between memory usage and runtime cost:

- With mostly-empty tables (high memory usage), collisions are expected to be rare (low runtime cost).
- With mostly-full tables (low memory usage), collisions are expected to be frequent (high runtime cost).

In practice, one typically *resizes* the hash table when it gets too full.

This requires a *family of hash functions* $h_N : \mathcal{K} \to \{0, \ldots, N-1\}$.

Let M be the *maximum size* of arrays in your system.
Let $h : \mathcal{K} \to \{0, \ldots, M-1\}$ be a hash function. One way to obtain h_N, $0 \leq N \leq M$, is via

$$h_N(i) = h(i) \bmod N.$$

Final notes on hash tables

Most dynamic hash tables are implemented on top of dynamic arrays using *chaining*.
Linear probing is especially usefull for *constant tables*.

Final notes on hash tables

Most dynamic hash tables are implemented on top of dynamic arrays using *chaining*.
Linear probing is especially usefull for *constant tables*.

	C++	Java
Set	`std::unordered_set` (C++11)	`java.util.HashSet`
Dictionary	`std::unordered_map` (C++11)	`java.util.HashMap`
Set (duplicates)	`std::unordered_multiset` (C++11)	
Dictionary (duplicates)	`std::unordered_multimap` (C++11)	

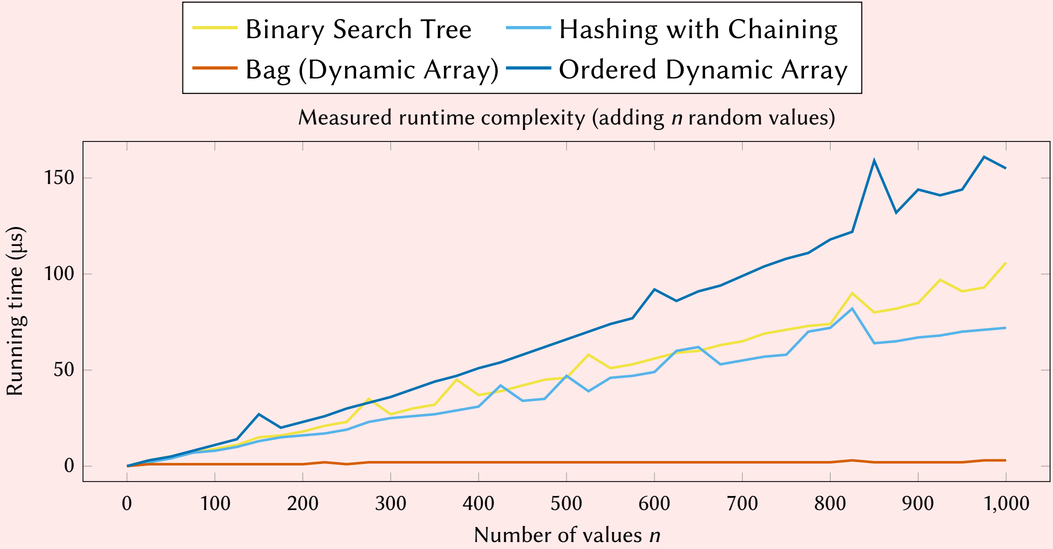
Sets and dictionaries in practice

	Cost	Ordered	Principle
Dynamic Arrays	$\Theta(N)$	No	
Ordered Dynamic Array[a]	$\Theta(\log_2(N))$, $\Theta(N)$	Yes	BinarySearch
Binary Search Trees	$\Theta(\log_2(N))$	Yes	Red-Black Trees.
Hash Tables	Expected $\Theta(1)$[b]	No	Chaining.

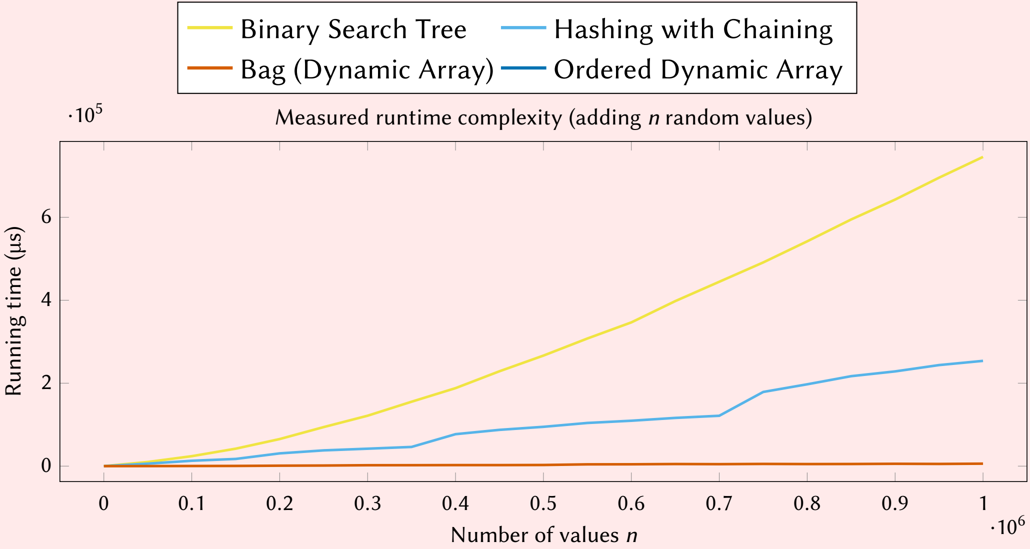
[a] Supported in C++23 via `std::flat_set` (set), `std::flat_map` (dictionary), `std::flat_multiset` (set, with duplicates), and `std::flat_multimap` (dictionary, with duplicates).

[b] For somewhat decent hash functions and large enough hash table.

Sets and dictionaries in practice



Sets and dictionaries in practice



Sets and dictionaries in practice